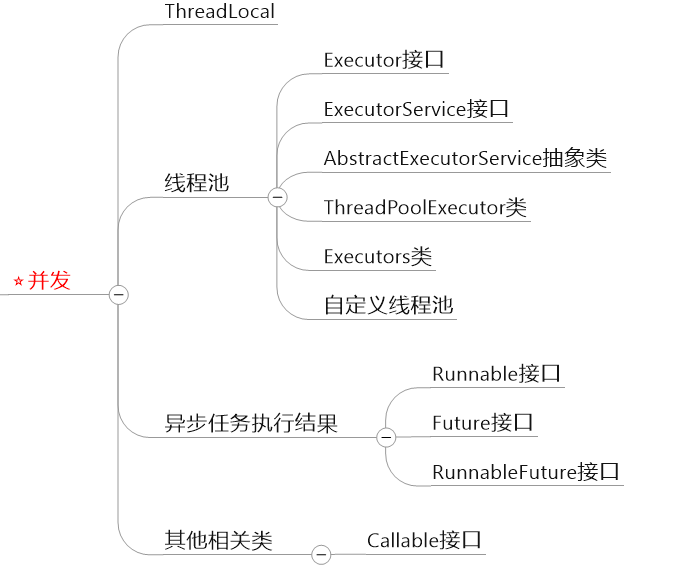
# 自定义线程池



## Executors类中工厂方法概述

查看Executors类的源码，能够发现Executors类中的工厂方法最终通过ThreadPoolExecutor类的构造函数创建线程池，例如：

public static ExecutorService newFixedThreadPool*(*int nThreads*) {* return new ThreadPoolExecutor*(*nThreads, nThreads,  
 0L, TimeUnit.*MILLISECONDS*,  
 new LinkedBlockingQueue*<*Runnable*>())*;  
*}*

public static ExecutorService newFixedThreadPool*(*int nThreads, ThreadFactory threadFactory*) {* return new ThreadPoolExecutor*(*nThreads, nThreads,  
 0L, TimeUnit.*MILLISECONDS*,  
 new LinkedBlockingQueue*<*Runnable*>()*,  
 threadFactory*)*;  
*}*

## ThreadPoolExecutor类构造函数概述

ThreadPoolExecutor类的构造函数总共有4个：

**Constructor：**

|  |
| --- |
| 1  public ThreadPoolExecutor*(*int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,BlockingQueue*<*Runnable*>* workQueue*)* |
| 2  public ThreadPoolExecutor*(*int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,BlockingQueue*<*Runnable*>* workQueue,  ThreadFactory threadFactory*)* |
| 3  public ThreadPoolExecutor*(*int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,BlockingQueue*<*Runnable*>* workQueue,  RejectedExecutionHandler handler*)* |
| 4  public ThreadPoolExecutor*(*int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,BlockingQueue*<*Runnable*>* workQueue,  ThreadFactory threadFactory,RejectedExecutionHandler handler*)* |

查看源码发现前3个构造函数依赖于第4个构造函数实现，先看看构造函数中各个参数作用：

**Parameters and Descriptions:**

|  |  |
| --- | --- |
| corePoolSize | the number of threads to keep in the pool, even if they are idle, unless allowCoreThreadTimeOut is set  除非设置了allowCoreThreadTimeOut参数，否则线程池中将维持corePoolSize个线程，即使有些线程是空闲的。 |
| maximumPoolSize | the maximum number of threads to allow in the pool  池中最多允许maximum个线程存在。 |
| keepAliveTime | when the number of threads is greater than the core, this is the maximum time that excess idle threads will wait for new tasks before terminating.  当池中的线程数超过了corePoolSize,线程的空闲时间不能超过keepAliveTime。 |
| unit | the time unit for the keepAliveTime argument  keepAliveTime参数的单位。 |
| workQueue | the queue to use for holding tasks before they are executed. This queue will hold only the Runnable tasks submitted by the execute method.  暂存待执行任务的队列，该队列只能暂存通过execute方法提交的任务。 |
| threadFactory | the factory to use when the executor creates a new thread.  executor方法通过threadFactory创建线程。 |
| handler | the handler to use when execution is blocked because the thread bounds and queue capacities are reached  因线程边界和队列容量问题而拒绝执行任务时的处理函数。 |

## 线程池主要处理流程



1. 客户端向线程池提交一个任务，如果此时池中线程数不足corePoolSize，则创建一个新的线程处理该任务（即使此时池中存在空闲线程）。
2. 如果池中线程数超过corePoolSize，则尝试将任务暂存到workQueue，如果暂存成功，待排在前面的任务开始执行且池中存在空闲线程，将分配一个空闲线程执行该任务。
3. 如果暂存失败，则进一步判断线程池中线程数是否达到maximumPoolSize，如果尚未达到，则创建一个新线程执行任务，如果已经达到，则拒绝执行该任务，并且启动设置的处理程序。
4. 如果线程池中线程数量超过corePoolSize，则清除空闲时间超过keepAliveTime的线程，直到线程中的线程数不大于corePoolSize为止。

以下为网上看到的一个例子：

举个通俗易懂的例子，公司要设立一个项目组来处理某些任务，hr部门给的人员编制是10个人（corePoolSize）。同时给他们专门设置了一间有15个座位（maximumPoolSize）的办公室。最开始的时候来了一个任务，就招聘一个人。就这样，一个一个的招聘，招满了十个人，不断有新的任务安排给这个项目组，每个人也在不停的接任务干活。不过后来任务越来越多，十个人无法处理完了。其他的任务就只能在走廊外面排队了。后来任务越来越多，走廊的排队队伍也挤不下。然后只好找找一些临时工来帮助完成任务。因为办公室只有15个座位，所以它们最多也就只能找5个临时工。可是任务依旧越来越多，根本处理不完，那没办法，这个项目组只好拒绝再接新任务。（拒绝的方式就是 Handler），最后任务渐渐的少了，大家都比较清闲了。所以就决定看大家表现，谁表现不好，谁就被清理出这个办公室（空闲时间超过 keepAliveTime），直到 办公室只剩下10个人（corePoolSize），维持固定的人员编制为止。

## 线程池其他配置

**预启动线程**

默认情况下，线程池中的核心线程只有当新任务提交才会创建和启动，但可以通过prestartCoreThread（）或prestartAllCoreThreads（）预先启动池中的线程。

**核心线程空闲时间**

如果当前池中的线程数超过corePoolSize，则多余的线程将在空闲时间超过keepAliveTime时终止。如果不想让核心池中的线程长时间空闲，可以使用allowCoreThreadTimeOut(boolean value)方法开启，keepAliveTime对核心池线程的约束。

更多的配置请查看官方文档：<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html>

## 参数设置技巧

配置线程池需要综合考虑很多问题：

1. **线程池任务性质**

计算密集型任务比较消耗CPU时间片，所以线程数一般设置为等于或略大于CPU的核数，如n+1。

IO密集型任务将时间主要花费在资源等待上，CPU压力并不大，所以可以多设置线程数，如2n+1。

1. **CPU使用率及CPU负载**

当线程数量设置过多：

* + - 1. 线程的初始化、切换和销毁将消耗不少资源，使CPU的使用率一直维持在较高水平。
      2. 接收到的任务迅速执行，短时间内给CPU带来较大压力。
      3. CPU使用过于集中，CPU使用率呈锯齿状，任务提交时CPU使用率短时间内飙升，任务执行完时CPU使用率又下降到很低的水平。

当线程数量设置过少：CPU负载之正在执行的线程和等待执行的线程之和，CPU负载过高说明，CPU的竞争较为激烈。

1. **下游系统抗并发的能力**

多线程给下游系统造成的并发等于你设置的线程数，例如多线程访问数据库，需要考虑数据库连接池大小的设置。如果访问的是下游系统的接口，就需要考虑下游系统能够承受怎样的并发量。

1. **服务器上运行的其他程序**